SANDIA REPORT

SAND2004-6428
Unlimited Release
Printed February 2005

SIERRA Framework Version 4: Solver Services

Alan B. Williams

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

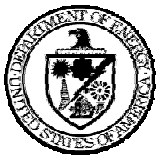
Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online



SAND2004-6428
Unlimited Release
Printed February 2005

SIERRA Framework Version 4: Solver Services

Alan B. Williams
Advanced Computational Mechanics Architectures Dept.
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0826

Abstract

Several SIERRA applications make use of third-party libraries to solve systems of linear and nonlinear equations, and to solve eigenproblems. The classes and interfaces in the SIERRA framework that provide linear system assembly services and access to solver libraries are collectively referred to as solver services. This paper provides an overview of SIERRA's solver services including the design goals that drove the development, and relationships and interactions among the various classes. The process of assembling and manipulating linear systems will be described, as well as access to solution methods and other operations.

Acknowledgement

The format of this report is based on information found in [17] and the example material at [2].

Contents

Appendix

Figures

SIERRA Framework Version 4: Solver Services

1 Introduction

Sparse systems of linear equations arise in several SIERRA applications, and the solution of linear systems is often the most computationally intensive portion of the application. The SIERRA framework provides services to assist with the assembly and manipulation of linear systems, and provides interfaces for accessing a number of third-party solver libraries. Services are also provided to assist in using nonlinear solvers and eigensolvers. This paper will give an overview of these services including design goals and descriptions of how to use the various classes and interfaces. However, specific details of method arguments and data types will be avoided in most cases as that information is best obtained from documentation generated directly from class header files. Developers are referred to the doxygen-generated documentation on the SIERRA web site [4, 3].

To set a context for the discussion that follows, we will first establish the notation for the linear systems that SIERRA's solver services are aimed at assembling and solving. Many implicit finite-element formulations, as well as finite-volume and others give rise to a linear system denoted by

$$Ku = f, \quad K \in \Re^{NxN}, \quad u, f \in \Re^{N} \tag{1}$$

where N is the number of degrees of freedom in the problem being solved, K is often referred to as the global "stiffness" or "system" matrix, f is referred to as the "load" vector, "forcing term" or more generally the "right-hand-side" and u is the solution being sought. (Note that linear systems are often denoted by $Ax = b$ in mathematical literature.) When an analysis includes constraints (e.g., due to contact and/or adaptive mesh refinement), it is often necessary to solve the linear problem subject to a system of constraint relations, denoted by

$$Cu = g, \quad C \in \Re^{N_c xN}, \quad g \in \Re^{N_c} \tag{2}$$

where N_c is the number of constraint relations. If constraints are imposed using a lagrange multiplier formulation, a logically partitioned linear system arises

$$\begin{bmatrix} K & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} u \\ \lambda \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \tag{3}$$

and notably, the matrix is indefinite which is an important consideration if choosing an iterative solution method. Alternative formulations for imposing constraints are available, including a penalty approach, as well as a form of static condensation which will be described in section 5.

An enormous body of literature exists on matrix computations and methods for the solution of linear systems. A few noteworthy examples are Golub and Van Loan [12], Saad [19], Freund and Nachtigal [11], and Saad and Schultz [20].

Applications written using the SIERRA framework depend on the framework for many services, including data management. The framework owns and manages solution data defined at mesh objects, etc. SIERRA's solver services provide infrastructure for taking data from the mesh database and using it to assemble linear systems. The application directs the linear system assembly process,

but the solver services manage the details, including making call-backs into application-provided algorithms, and passing data into solver-library data structures, etc. The purpose of this document is to describe the classes and components that make up SIERRA's solver services.

1.1 Organization of Paper

In section 2 the design goals that drove the development of SIERRA's solver services will be described. Section 3 describes the relationships and roles of the major classes that provide solver services. Section 4 describes linear system assembly and manipulation, and section 5 provides a brief review of the Finite Element Interface to Linear Solvers (FEI). Section 6 covers available mathematical operations (such as system solution, residual calculations, etc.). Section 7 covers miscellaneous topics such as using multiple linear systems, and section 8 describes the interfaces to nonlinear solvers and eigensolvers.

2 Design Goals

The benefits of using the solver services and abstraction layers provided by the framework are described in the context of the design goals in the following subsections. Strictly speaking, SIERRA applications could use solver libraries directly, without the help of framework solver services. However, it is hoped that the advantages provided by the solver services make that an unattractive option.

2.1 Solver-library Abstraction

Depending on the spectrum of problems addressed by an application, there may be no single solver package capable of solving all of the linear systems that arise. For this reason, a major design goal for solver services in SIERRA has been to provide access to a large number of different solver libraries, and to allow applications to easily switch between them. Applications may switch from one solver library to another at run-time, i.e., without altering any application code or recompiling. Due to the wide variety of interfaces and data formats associated with different libraries, it is necessary to insert abstraction layers between the applications and the solver libraries in order to support this design goal. Some aspects of the abstraction are provided by the SIERRA framework and others by the Finite Element Interface to Linear Solvers (FEI).

2.2 Mapping Degrees-of-freedom to Algebraic Equations

Another design goal (which arises in part due to the goal of allowing easy switching of solver libraries) is to allow applications to address matrices and vectors (specify locations) using mesh-object identifiers and field- variables (e.g. "temperature field on node 9876"), rather than using algebraic equation numbers and indices. In other words, applications are relieved of the task of mapping degrees-of-freedom to a globally consistent algebraic equation space. This is more significant for

analyses which use multiple fields and different types of mesh-objects (e.g. nodes, edges, etc.) than it is for "topologically simple" analyses which solve a single scalar field at each node.

2.3 Parallel Communications

A third major design goal is to insulate applications from some of the parallel communications issues that arise when assembling data into matrices and vectors in a distributed-memory multi-processor setting. For example, when assembling data for finite-element nodes that are shared among multiple processors, the application may contribute each node's data on the local processor and the data is automatically sent to the processor that uniquely owns the corresponding equations in the linear system.

3 The Big Picture

Solver services in the SIERRA framework are provided by a small number of significant components or entities:

- **`Fmwk::LinearSystem`** This is the class which orchestrates the creation and assembly of linear systems. This class does most of the translation of data from SIERRA objects such as regions, mesh-object registrars and field variables, into the form needed by the FEI for assembly into the underlying linear system.
- **`Fmwk::Solver_Support`** Implementations of this abstract interface provide factories for instantiating FEI objects and coupling them with solver-library-specific data objects, as well as uniform interfaces for accessing library-specific solution methods, etc.
- **FEI** This subsystem provides uniform interfaces (i.e., independent of which solver library is being used) for creating and manipulating linear system data objects.
- **Solver-Libraries** Generally referred to as "third-party" libraries, they provide the actual data structures and solution methods for working with and solving linear systems.

Figure 1 shows the major classes and libraries which make up the solver services in the SIERRA framework. The dependence arrows indicate "uses" or "accesses" relationships. The abstraction layer which facilitates solver-library independence, is comprised of `Fmwk::SolverSupport` and the FEI. The implementation code for SIERRA applications, as well as `Fmwk::LinearSystem`, can be completely independent of the run-time type of underlying linear system objects.

`Fmwk::Solver_Support` is an abstract class (contains pure virtual methods). There is a separate implementation of `Fmwk::Solver_Support` for each solver library. At run-time, when `Fmwk::LinearSystem` is constructed, it is matched with the appropriate implementation of `Fmwk::Solver_Support` according to input-file parameters specifying which solver is to be used. During the creation and assembly of linear systems, `Fmwk::LinearSystem` can request instances of data objects such as matrices and vectors, which are created by factories that originate in a library-specific

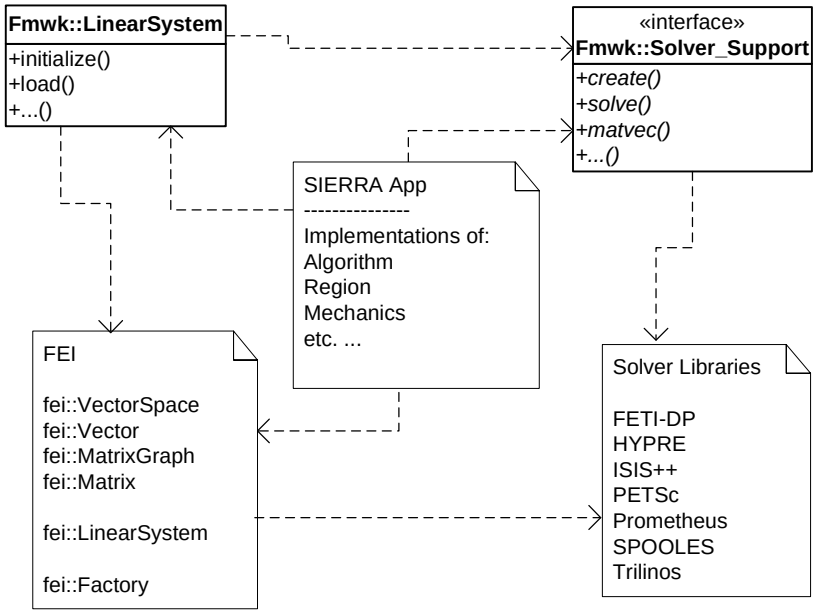


Figure 1. Major solver services components.

`Fmwk::Solver_Support` implementation. These data objects are contained in generic FEI interfaces which serve as thin containers to be passed through various library-independent code-scopes. The FEI interfaces also provide generic data input/output and mechanisms for manipulating the underlying data objects.

4 Assembly of Linear Systems

This section describes the classes and interfaces used to assemble linear systems. Usually applications assemble linear systems as a unit that includes a matrix and two vectors (right-hand-side and solution) but it is also possible to obtain and work with individual matrix and vector objects. The linear system assembly services support assembly from, and return solutions for, any scalar or vector field associated with any of the mesh-object types supported by the framework (e.g., nodes, edges, faces, etc.). Depending on the physics and the formulation involved, the mapping of degrees of freedom to a globally consistent algebraic equation space can range from trivial to very complex. Linear system assembly includes defining sets of indices that induce vector-spaces, interactions and connectivities that define matrix-graphs, and contribution/manipulation of coefficients in the matrix and vector objects.

Consider the trivial 5-element mesh of 2D quads in figure 2. Assuming a finite-element application with 1 scalar degree of freedom per node, the element stiffness matrices are of size 4x4, and the assembled matrix has coefficients in the positions marked by 'X' in the figure. This example is trivially simple in order to illustrate several points in the process of linear system assembly. In

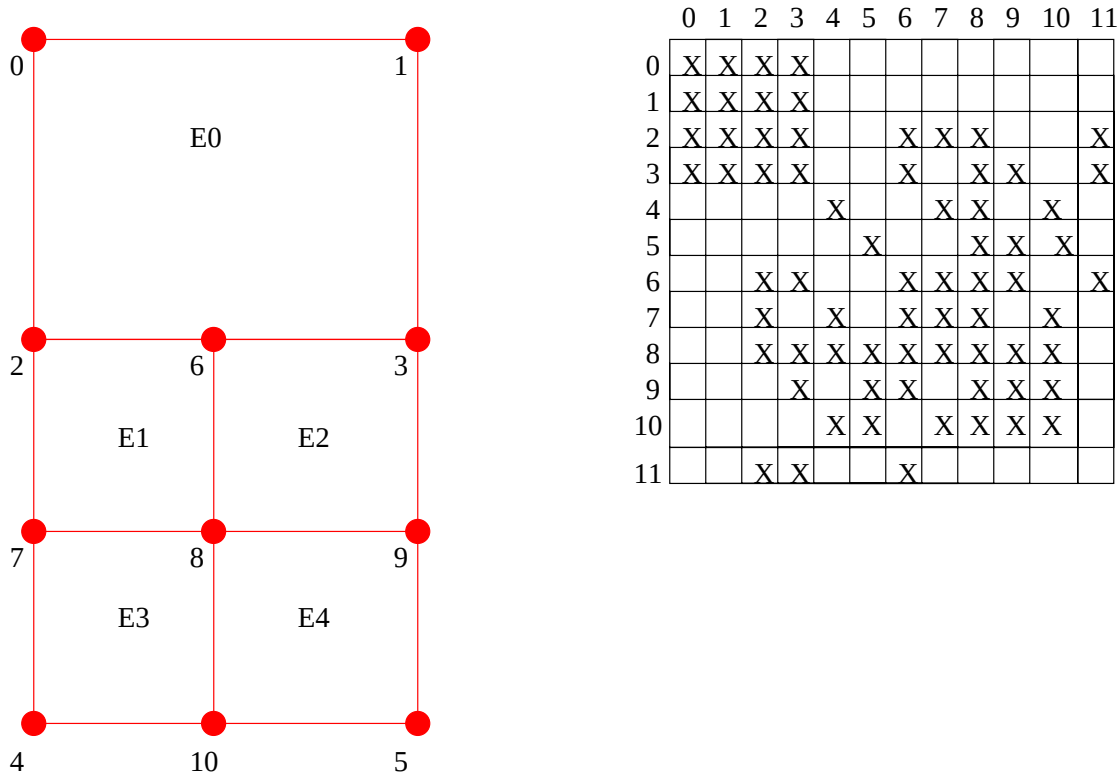


Figure 2. Trivial mesh, and matrix graph.

general, we are constructing sparse matrices from unstructured meshes. The creation of a sparse matrix in SIERRA takes a specific sequence as follows.

1. Accumulate structural data (e.g., element-node connectivities) that defines the matrix-graph (the locations of the nonzero coefficients in the matrix).

2. Allocate memory (depending on the underlying solver library being used – most require pre-allocation of matrix storage) for the matrix.

3. Load the coefficient data into the matrix.

This sequence requires making two distinct passes over the mesh: once to obtain the structural data, and again to obtain the associated coefficient data. The reason for this is that many solver libraries store the matrix coefficients in a single “flat” array, and the size of the array can’t be known until after the matrix-graph is fully defined.

In the trivial example of figure 2, it is easy to see that the structure of the matrix consists of blocks that correspond to the elements in the mesh. The nodal connectivity list for element “E0” is [0,1,2,3], which translates to the structurally symmetric 4x4 region of the matrix in rows 0 to 3 and columns 0 to 3, where the coefficients for that element’s stiffness matrix will be stored. Node 6 is referred to as a “hanging” or “mid-side” node, and we impose a constraint-relation to ensure that the solution at that node lies on a “straight line” between the solutions at nodes 2 and 3. Note that the

matrix-graph shown has coefficients in row 11 and column 11. These are the positions occupied by constraint-weights if a lagrange multiplier constraint is used to constrain node 6, and the result is an example of the partitioned matrix in equation (3). If this constraint were imposed using a penalty formulation instead of a lagrange multiplier formulation, then the matrix would only have 11 rows and columns, instead of the 12 that are shown.

Assembling and accessing a linear system involves a number of classes and interfaces which are listed here and then described individually in more detail in the following subsections.

- `Fmwk::LinearSystem`: This is the primary class that an application interacts with to coordinate the initialization and assembly of a linear system.
- `Fmwk::AlgorithmLinSys` interfaces: This is a family of four interfaces (abstract classes) that define call-back mechanisms for providing linear system contributions and constraint definitions. Application-implementations of these interfaces can be registered with the `Fmwk::LinearSystem` class, which then calls methods on them as a way of requesting data at appropriate points in the linear system assembly process. This leaves the application in charge of looping over mesh-objects for which contributions are to be made, etc.
- `Fmwk::LinearSystem::BCSet` interface: This is an interface which can be implemented by applications in order to define dirichlet or essential boundary condition sets, i.e., sets of mesh-objects for which boundary conditions are to be imposed. Like the AlgorithmLinSys interfaces, this is a call-back mechanism whereby the `Fmwk::LinearSystem` class can request the boundary condition data from the application at the appropriate point in the linear system assembly.
- `Fmwk::LinearSystem::GeneralBCSet` interface: This is very similar to the BCSet interface described above, but is better suited to applying boundary conditions to vector fields. The BCSet interface is the best choice if scalar fields are being used.

During and after the assembly process, the `Fmwk::LinearSystem` class holds the matrix-graph, matrix and vector objects in containers that may be accessed by the application (these containers are FEI objects, which will be described in section 5). This allows the application to control the process of calculating residuals, passing the linear system to a solver, etc.

4.1 `Fmwk::LinearSystem` Overview

Usage of the `Fmwk::LinearSystem` class can be divided into a number of distinct phases:

1. Construction: In addition to creating a `Fmwk::LinearSystem` instance, the construction phase also includes registration of algorithms for contributions, constraints and boundary condition sets. Note that implementations of the `AlgorithmLinSys*` interfaces are registered with the `Fmwk::LinearSystem` instance when they are constructed. This is done by the constructor for each of those base classes, no explicit action is needed by the derived class.
2. Initialization: The method `Fmwk::LinearSystem::initialize()` performs the initialization of vector-space and matrix-graph objects. During the execution of this method, call-backs are made to application implementations of the `AlgorithmLinSys*` interfaces, etc.

3. Loading: This phase involves the loading of coefficient data into the linear system (again, via call-backs to application implementations of `AlgorithmLinSys*` interfaces), enforcement of boundary conditions, etc.

In addition to these phases, there are also a variety of methods for accessing and manipulating the linear system objects, returning solution data to the associated `Fmwk::Region`, etc.

Following are the methods on the `Fmwk::LinearSystem` class which are relevant to application usage. They are listed in approximately the order in which they would be called by a typical application, except for the query methods listed at the end, which may be called almost any time.

- **`register_bc_set()`** Registers implementations of the `Fmwk::LinearSystem::BCSet` interface, which will be described in section 4.7.
- **`initialize()`** Orchestrates the initialization of the linear system's matrix graph and vector space objects, by calling the init() methods on the registered algorithm instances to define connectivity lists for contributions and constraints.
- **`load(), load_contributions(), load_constraints(), load_bc(), load_complete()`** The load() method internally calls load_algorithms(), load_constraints(), load_bc() and load_complete() in that order. Alternatively, applications may make those individual method calls themselves rather than calling the load() method.
- **`set_initial_guess(),set_rhs()`** These methods load specified data into the linear system's solution vector or right-hand-side vector, respectively.
- **`zero_matrix_rows()`**
- **`load_complete()`** Signals that all coefficient data has been loaded into the linear system. At this point the underlying library may finish assembling the linear system, by completing boundary condition enforcement, communicating shared data to the appropriate processors, etc.
- **`solve()`** There are several overloadings of the solve() method, some of which have output arguments such as initial and final residual norms, number of iterations taken, etc. These are helper methods which call through to the solve and residual-calculation methods on the `Fmwk::Solver_Support` interface, which will be described in section 6.
- **`residual_norm()`** Like the solve() methods, this method calls through to the corresponding method on the `Fmwk::Solver_Support` interface.
- **`scatter(),get_lagrange_multipliers()`** These methods are for returning solution data after the linear system has been solved. The scatter() method copies data from the solution vector to the owning `Fmwk::Region` class, while the get_lagrange_multipliers() method returns the lagrange multiplier solutions directly to the caller in a container of double precision values.
- **`copy_vector_to_region(),copy_region_to_vector()`** These methods provide functionality that can be equivalent to what is done by the scatter() and set_initial_guess() methods. The distinction is that these methods take an argument specifying which vector is to be copied to/from, rather than automatically using the linear system's solution vector.

- **reset(),reset_rhs(),reset_cr()** These methods set zeros throughout the linear system's matrix and vectors. reset_cr(), however, is a special case. It attempts to reset only the portion of the linear system associated with constraints. This operation is not supported by all solver libraries.

- **deallocate()** Destroys the internal linear system components, such as matrix-graph, matrices and vectors, etc.

- **get_matrix_graph(), set_matrix_graph(), get_linear_system(), set_linear_system(), get_fei_factory()** These methods are used to share or swap components of the linear system among different instances of `Fmwk::LinearSystem`. Note that the components of the linear system don't exist until after intialize() and/or the load methods have been called, depending on which component is being requested.

- **region()** Query for the `Fmwk::Region` object with which this `Fmwk::LinearSystem` instance is associated.

- **elapsed_cpu_time(), elapsed_wall_time(), reset_cpu_timer(), reset_wall_timer()** Timer access methods are for measuring the amount of time spent in various `Fmwk::LinearSystem` methods.

4.2 `Fmwk::AlgorithmLinSys` Interfaces Overview

The "AlgorithmLinSys" interfaces are a set of four interfaces for making coefficient contributions (e.g., element stiffness and loads) and for defining constraint relations. As an example, nodal finite-element applications need to define a matrix-graph structure using element-node connectivity lists, and contribute coefficients for element-stiffnesses and load vectors. These applications define an algorithm which implements the appropriate one of these interfaces and which can loop over the elements and produce the connectivity lists and coefficient contributions when directed to do so via calls to the methods defined on the interface. Note that the SIERRA framework provides an implementation (`Fmwk::LinSysWorksetAlgorithm`) that makes connectivity and coefficient contributions, given specified workset variables. This is created as a nested algorithm for application-implementations of `WorksetAlgorithm` which are to make linear-system contributions. Figure 3 depicts the four AlgorithmLinSys interfaces, along with the framework implementation `LinSysWorksetAlgorithm`, and shows their inheritance relationship to `WorksetAlgorithm` and `Algorithm`.

The four interfaces are:

- **Fmwk::AlgorithmLinSysRegularContribution** Blocks of self-homogeneous contributions. For example, a block of element-contributions for elements having the same topology and field layout. Or, an algorithm that strides over some set of mesh-objects and makes any contribution that conforms to the same stencil each time.

- **Fmwk::AlgorithmLinSysArbitraryContribution** Blocks of contributions that may vary in size and shape from one to the next, i.e., using a different stencil for each contribution. (The LinSysStencil class is described in the next section.)

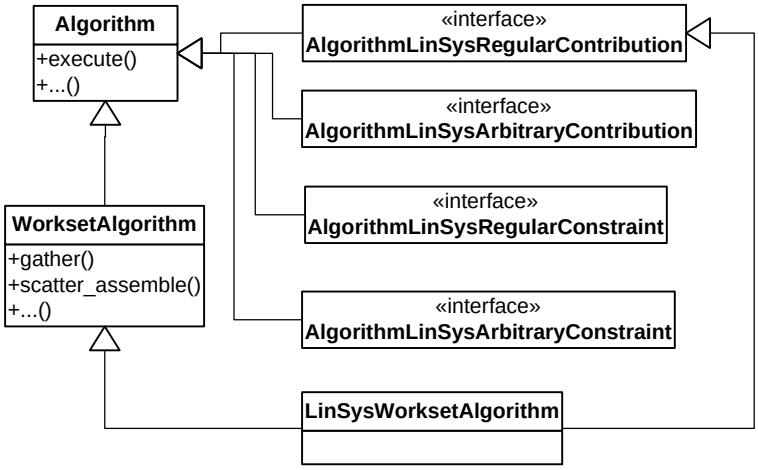


Figure 3. AlgorithmLinSys interfaces inherit Algorithm

- **`Fmwk::AlgorithmLinSysRegularConstraint`** Blocks of constraint relations which are self-homogeneous. In other words, constraints which each constrain the same field on a fixed number of the same types of mesh-object.

- **`Fmwk::AlgorithmLinSysArbitraryConstraint`** Blocks of constraint relations for which the connectivity list may change size and the constrained field may change from one constraint to the next.

As mentioned earlier, `Fmwk::LinearSystem` loops over registered instances of these interfaces at the appropriate point in linear system assembly, and directs them to provide their contributions. The pattern used is a double call-back approach which works as follows. Each of these interfaces declares an init() method and an apply() method, which are pure-virtual methods to be implemented by the application derived class. The init() method will be called from `Fmwk::LinearSystem::initialize()`, at which point the implementation is expected to stride through its set of contributions and provide each connectivity list via calls to the init_connectivity() method. The init_connectivity() method is a non-virtual method on the algorithm interface, and it relays the connectivity list to `Fmwk::LinearSystem`. During the course of the linear system initialization, this process will be repeated for each algorithm that is registered on the `Fmwk::LinearSystem` instance. Later during the coefficient loading stage, the apply() method will be called from one of the `Fmwk::LinearSystem` load methods (either `Fmwk::LinearSystem::load_contributions()` or `Fmwk::LinearSystem::load_constraints()` as appropriate). At this point the implementation is expected to stride through its set of contributions again, this time providing coefficients by calling the apply_coefficients() method. The apply_coefficients() method is another non-virtual method on the algorithm interface which relays the data to the `Fmwk::LinearSystem` instance. The methods declared by these interfaces are listed here.

- **`init()`** This method is pure virtual, must be provided by implementations of the interface. This method will be called from within the `Fmwk::LinearSystem` class.

- **`init_connectivity()`** (Takes an argument which is an array of mesh-objects.) This method is non-virtual (implementation is provided by the SIERRA framework). It is expected to be called repeatedly from within the init() method as the implementation loops over the mesh extent for which contributions are being made. Each of the connectivity lists are relayed to the `Fmwk::LinearSystem` object by this method.

- **`apply()`** This method is pure virtual, must be provided by implementations of the interface. This method will be called from within the `Fmwk::LinearSystem` class.

- **`apply_coefficients()`** (Takes arguments which are coefficient arrays.) This method is non-virtual (implementation is provided by the SIERRA framework). It is expected to be called repeatedly from within the apply() method as the implementation loops over the mesh extent and produces coefficients to be contributed. The implementation must loop over the mesh extent in the same order now as it did in the init() method. Each of the coefficient contributions are relayed to the `Fmwk::LinearSystem` object by this method.

4.3 Linear System Stencils

The class `Fmwk::LinSysStencil` is used as an argument in several of the `Fmwk::Algorithm-LinSys` constructors and methods. It provides a mechanism for mapping degrees of freedom to contribution coefficients. LinSysStencil is a specialization of the C++ standard library vector, and contains pairs. Each pair contains an ordinal and a pointer to a field argument. For the example of a block of element-contributions, the stencil would list the ordinals of connected mesh-objects (e.g. nodes) and pair them with associated nodal fields. This allows the LinearSystem object to unpack the coefficients and map them to the correct global indices in the matrix and/or vector.

Example 1.: Consider the solution of a single nodal scalar field 'u' with contributions from linear hexahedral elements. In this case the stencil would be

$$\{(0,u),(1,u),(2,u),(3,u),(4,u),(5,u),(6,u),(7,u)\}$$

where the first coordinate is the local node ordinal (e.g., offset into the element's connected nodes) and the second coordinate is a reference to the field variable.

Example 2.: Consider a 2D mesh with triangular elements and two fields u and v. Group by nodes:

$$\{(0,u),(0,v),(1,u),(1,v),(2,u),(2,v)\}$$

Group by fields:

$$\{(0,u),(1,u),(2,u),(0,v),(1,v),(2,v)\}$$

LinSysStencils may become very complex with multiple fields or contributions from more than one type of mesh-object (e.g. nodes, edges, etc.). Note that the type of the mesh-objects is not explicitly held in the LinSysStencil, except via the association of the fields.

4.4 Regular Contributions

Regular contributions are homogeneous within each instance of the `Fmwk::AlgorithmLinSys-RegularContribution` interface. An example is a block of elements in a finite-element analysis

which all have the same topology and layout of field variables. Each element is a contribution (connectivity list and stiffness/load coefficients) and the extent of the algorithm instance is limited to the block of similar elements. The size and layout of each contribution's degrees of freedom are described using a `Fmwk::LinSysStencil` argument when the algorithm instance is constructed, and that stencil then applies to all contributions made by the algorithm instance.

4.5 Arbitrary Contributions

Arbitrary contributions are not required to be homogeneous, each individual contribution may be a different size and shape. For this reason, no `Fmwk::LinSysStencil` argument is provided when an implementation of `Fmwk::AlgorithmLinSysArbitraryContribution` is constructed, but two `Fmwk::LinSysStencil` arguments are provided with each contribution. One stencil describes the contribution's layout in the row dimension, the other describes its layout in the column dimension.

4.6 Regular Constraints and Arbitrary Constraints

The distinction between regular and arbitrary constraints is the same as the distinction between regular and arbitrary contributions. Regular constraints are homogeneous within the entire block of constraints, while arbitrary constraints may have different numbers of constrained mesh-objects and different constrained fields for each constraint. The names of these two interfaces are `AlgorithmLinSysRegularConstraint` and `AlgorithmLinSysArbitraryConstraint` (both are in the `Fmwk` namespace).

Each constraint is defined by a list of constrained mesh-objects, the field to be constrained at each mesh object, a list of coefficient weights and a right-hand-side value. If the solution field at the i-th constrained mesh object is denoted by u_i, and the i-th coefficient weight is denoted by w_i then the constraint enforces the relationship

$$u_0 w_0 + u_1 w_1 + \cdots + u_n w_n = rhsvalue \tag{4}$$

where n is the number of mesh-objects in the constraint.

If a penalty formulation is used to enforce the constraint, then new nonzeros may or may not be added to the matrix-graph, depending on whether couplings among the mesh-objects in the constraint already exist due to other interactions in the analysis. The contribution that is made to the linear system for a penalty constraint is described by the following pseudo-code. Let the matrix be denoted by A, and the right-hand-side by b. Note that the array "`index`" contains mappings from the constrained mesh-objects and fields, to the corresponding indices in the global equation space.

```
for(i=0; i<n; ++i) {
  b[index[i]] += w[i] * rhsvalue * penaltyvalue;

  for(j=0; j<n; ++j) {
    A[index[i],index[j]] += w[i] * w[j] * penaltyvalue;
  }
}
```

If a lagrange multiplier formulation is used, then a row and column is added to the global matrix for each constraint. In a distributed-memory parallel setting, constraint rows and columns are appended to the local portion of the global matrix on each processor.

4.7 Enforcement of Boundary Conditions

Enforcement of essential or Dirichlet boundary conditions is accomplished via the interfaces `Fmwk::LinearSystem::BCSet` and `Fmwk::LinearSystem::GeneralBCSet`. These are abstract interfaces which are expected to be implemented by applications. Instances of these interfaces are registered with the `Fmwk::LinearSystem` class, which later calls methods on these instances to request the boundary condition data from the application at the appropriate time during the assembly of the linear system.

The `Fmwk::LinearSystem::BCSet` interface defines methods for passing the following information to specify boundary conditions.

- **mesh-object type** Whether the boundary condition is being applied to nodes, edges, etc.
- **variable** The field for which a value is being prescribed.
- **size** Number of mesh-objects in the boundary condition.
- **identifiers** Array of length 'size', of the mesh-object identifiers in the boundary condition.
- **values** Array of length 'size' of coefficient values being prescribed.

For cases where boundary conditions need to be imposed on more than one field or on more than one type of mesh-object, separate instances of the interface need to be used.

The `Fmwk::LinearSystem::GeneralBCSet` interface is very similar to the BCSet interface, but is better suited to applying boundary conditions to vector fields. The differences are aimed at allowing values to be prescribed for more than one component of a vector field, or for different components of the field on each mesh-object in the boundary condition set.

Essential boundary conditions are enforced in the linear system as follows. The corresponding rows and columns in the global matrix are zeroed and 1's are placed on the diagonal. (Before a column is zeroed, its coefficients are multiplied by the boundary condition's prescribed value and subtracted into the appropriate positions in the right-hand-side.) Then, the prescribed values are placed in the right-hand-side. Naturally this can result in the boundary condition being enforced only as accurately as the tolerance which was set on the solver. If the user wishes to ensure exact boundary condition enforcement they can specify that an alternate approach is taken whereby zeros are placed in appropriate positions in the right-hand-side before the system is solved, and the prescribed values are explicitly placed in the solution vector afterwards. (This is specified at run-time, by placing the line-command **'bc enforcement = exact'** in the solver-block of the input file.)

5 Finite Element Interface to Linear Solvers

The Finite Element Interface to Linear Solvers (FEI) provides an abstraction layer for assembling linear systems (see [22]). The FEI is the primary mechanism through which `Fmwk::LinearSystem` creates and fills matrices and vectors, as the FEI abstraction allows the code in `Fmwk::LinearSystem` to be independent of the solver-library being used. Originally the FEI's interface was expressed using a single class (called "FEI") and had a heavy orientation towards nodal finite-element formulations (see [7]). Recently, the FEI has been augmented with a group of several smaller interfaces in order to make it more flexible and more general. These interfaces now reside in a namespace called 'fei::'. The original interface is still maintained however, to ease the process of switching to the new ones. The new interfaces share some implementation code with the old interface, but there is also a lot of new code. When the new interfaces were initially phased in, there were some use cases handled by the original FEI that the new interfaces couldn't handle. For this reason we retained the ability to choose between the old and new fei at run-time. The new interfaces are used by default for most cases at the time of this writing, but the user may choose the original FEI by using the command **'select fei = old'** in the solver-block of their input file. One of the reasons for adding the new interfaces was to allow the assembly of data from arbitrary mesh-object types, rather than just nodes. So in some cases, if the user attempts to force the use of the old FEI it will cause an error. Additionally, the new implementation code substantially improves the performance of matrix assembly, so the intent is to completely switch to the new code and phase the old code out entirely.

In SIERRA the fei interfaces are predominantly used by `Fmwk::LinearSystem` for assembling data into matrices and vectors, and as containers for passing matrix and vector arguments to and from `Fmwk::Solver_Support` methods. They may also be used directly by applications in circumstances where specific operations are not supported by Fmwk::LinearSystem and other SIERRA framework classes.

Classes in the `fei::` namespace are all abstract interfaces, with implementations that reside in a `snl_fei::` namespace. The following interfaces are members of the fei:: namespace.

- **`VectorSpace`** Maps sets of degrees of freedom to a globally consistent algebraic equation space. In SIERRA a degree of freedom is fully specified by a mesh-object identifier and an associated field (and an offset into the field if it is a vector field). The same concepts exist in the fei interfaces but with slightly different terminology. SIERRA enumerates the various mesh object types with `Fmwk::MeshObj::DerivedType`, while fei has integer identifier-types which are user-specified. In SIERRA a field is described by `Fmwk::Field`, while fei has integer field-identifiers which are user-specifiable in terms of size (number of scalar components), etc. One of the tasks performed by a `Fmwk::LinearSystem` instance is to establish mappings between `Fmwk::MeshObj::DerivedTypes` and fei identifier-types, and to map the relevant `Fmwk::Fields` to a set of fei field-identifiers with associated sizes, etc. `fei::VectorSpace` can answer queries such as: given a particular mesh-object identifier (e.g. node 9876) and associated field (e.g. temperature), return the corresponding global equation number. fei equation numbers are globally zero-based.

- **`MatrixGraph`** Accumulates connectivity lists and generates an algebraic matrix graph. Connectivity lists may include element-to-node connectivities, constraint-relation connectivities, etc.

- **`Vector`** Thin container that provides an abstraction layer for library-specific vector objects. The abstract `fei::Vector` class is implemented by `snl_fei::Vector`, which is a template. `snl_fei::Vector` is templated on the type of the underlying library-specific vector. Thus, it is easy to get the underlying vector from `fei::Vector` using dynamic_cast in code scopes that know which type to cast to.

- **`Matrix`** Thin container that provides an abstraction layer for library-specific matrix objects. Like `fei::Vector`, `fei::Matrix` is implemented by a template in the `snl_fei::` namespace.

- **`LinearSystem`** Container that binds a matrix and two vectors (solution and right-hand-side) for the purposes of essential boundary condition enforcement, etc. Also convenient for passing a linear system as a single argument between various code scopes.

- **`Factory`** The run-time type of instances of this interface vary depending on the underlying linear algebra library being used. Its purpose is to produce new instances of objects such as `fei::Matrix` and `fei::Vector`, the run-time type of which is consistent with other objects already in use.

5.1 Constraint Reduction

One of the data-filtering services provided by the FEI is the removal of "master-slave" constraints during linear system assembly. As mentioned in section 1, solution of a linear system subject to constraint relations can require solving the partitioned system shown in equation (3). The matrix in equation (3) is indefinite, and can be difficult to solve or precondition effectively.

The approach used for projecting the constrained system into a reduced space is described in a paper by Saint-Georges et al [21] and is briefly summarized here.

If the constraints represent master-slave relations (one degree of freedom is slaved to a linear-combination of other degrees of freedom), the constraint matrix C from equation (2) can be expressed as

$$C = \begin{bmatrix} D & -I \end{bmatrix} \tag{5}$$

and D is referred to as the dependency matrix. The solution vector u can be split into dependent and independent unknowns and written as an expression involving D,

$$u_d = Du_i + g \tag{6}$$

and the global stiffness matrix K can be partitioned according to dependent and independent variables as follows.

$$K = \begin{bmatrix} K_{ii} & K_{id} \\ K_{di} & K_{dd} \end{bmatrix} \tag{7}$$

Then a reduced matrix K_r of size $N - N_c$ (N degrees-of-freedom, N_c constraints) is given by

$$K_r = K_{ii} + K_{id}D + D^T K_{di} + D^T K_{dd} D \tag{8}$$

and if K is symmetric and positive definite, then so is K_r. A reduced right-hand-side f_r is given by

$$f_r = f_i + D^T f_d \tag{9}$$

and the problem of solving equation (1) subject to equation (2) is equivalent to solving the reduced system

$$K_r u_i = f_r \tag{10}$$

which is usually much easier to solve with an iterative method. In some cases SIERRA applications have been able to solve the reduced system when the unreduced system couldn't be solved. The reduction can be carried out by the FEI using local operations during element-wise assembly of the linear system. It is completely transparent to the user, and the solution data is returned in the original unreduced space.

6 Mathematical Operations

Once a matrix and/or vectors have been assembled, operations may be performed such as calculating residual norms

$$||r||_p, \quad r = b - Ax \tag{11}$$

(where p is user-selectable), as well as matrix-vector products and linear system solutions.

These types of operations generally involve library-specific code, as each library has a different interface for constructing and/or accessing solver and preconditioner objects and other methods. This is where the abstraction layer provided by `Fmwk::Solver_Support` is utilized. The run-time type of the `Fmwk::Solver_Support` implementation need not be known to the application, but the implementation of its methods must be library-specific.

6.1 `Fmwk::Solver_Support`

`Fmwk::Solver_Support` is a class which defines methods for interacting with the underlying solver library to perform tasks such as system solutions, residual calculations, etc. There are several implementations of this interface provided by the framework, one for each supported solver library (see figure 4). Thus, the run-time type of the `Fmwk::Solver_Support` instance associated with the `Fmwk::LinearSystem` object is specific to the solver library being used. The matrix, vector and linear system arguments passed to and from the methods on this class are generic FEI objects, which constitute thin containers from which the library-specific `Fmwk::Solver_Support` implementation may easily extract the underlying library-specific data object. The methods currently defined by this interface which are relevant to application needs are:

- **`solver_type_name()`** Returns the name of the underlying solver library, such as "Trilinos", "PETSc", etc.
- **`solver_method_name()`** Requires an argument of type `Fmwk::Parameters`. Returns a name describing the solution method specified in the input-file solver-block that produced the contents of the `Fmwk::Parameters` object. Example: "Aztec AZ_gmres AZ_jacobi".
- **`is_option_set(),set_option()`** Both of these methods require `Fmwk::Parameters` arguments. They query and set, respectively, the specified option.

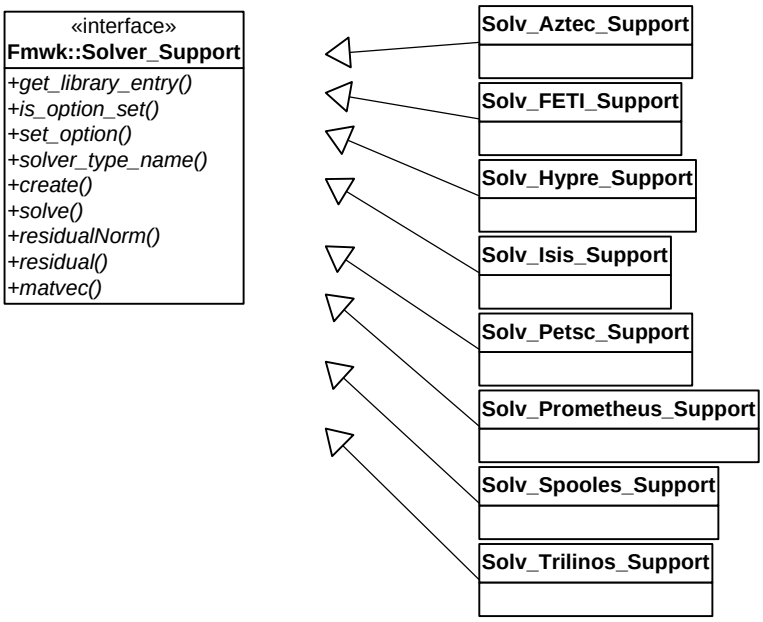


Figure 4. Library-specific implementations of `Fmwk::Solver_Support`.

- **`solve()`** Requires arguments including containers holding the linear system, specified solver-control parameters, as well as an optional preconditioning matrix. In general, iterative solvers solve the preconditioned system

$$M_1^{-1} A M_2^{-1} y = M_1^{-1} b, \quad y = M_2 x \tag{12}$$

  where the preconditioning matrix is denoted by $M = M_1 M_2$ and M is some approximation to A. The inversion and/or splitting of the preconditioning matrix, when performed, is typically done internally by the solution method and is transparent to the user. In many cases the preconditioning matrix M is calculated from A, and this calculation is done internally by the solver. But SIERRA allows for the case where the user assembles and provides a separate preconditioning matrix.

- **`residualNorm(), residual()`** `residualNorm()` calculates the norm $||r||_p$ (with p specified by the caller), while `residual()` calculates the vector r.

- **`matvec()`** There are a couple of overloadings of the `matvec()` method, which take an `fei::Matrix object`, and vector arguments as `fei::Vector` objects or as raw coefficient arrays.

6.2 Solver Option Parsing

The various implementations of the `Fmwk::Solver_Support` class in figure 4 play another role, which is the parsing of options and control parameters to be passed to the underlying library.

When a SIERRA application is run, a solver-block in the input file specifies which solver-library is to be used, and also specifies control parameters such as the solution method to be employed (e.g., CG versus GMRES, etc.), as well as convergence tolerance, iteration limits, etc. Naturally each library has not only a different set of possible options, but also a different method for setting them. Some libraries accept strings while others accept integers, etc.

The SIERRA parser subsystem converts the contents of the input file into a combination of enumerations and other values (including floating-point numbers, strings, etc.) and these are given to the parser-handlers in the various `Fmwk::Solver_Support` implementations such as `Solv_Aztec-_Support`. These library-specific parser-handlers are then responsible for mapping these values to library-specific values. This process is mostly transparent to SIERRA users as well as application code-developers, but it is worth describing in order to avoid confusion due to the way a "singleton" pattern is employed in combination with static methods, etc.

The solver-support implementations are singletons which, upon construction, install parser support for the use of input-file solver-blocks that specify a given library. For example, the `Solv_Aztec-_Support` singleton installs parser-support for the Aztec library. The methods that parse Aztec options and control parameters are static methods. These methods add the resulting parameters to instances of the `Fmwk::Parameters` class. Note that an input file may contain multiple distinct Aztec solver-blocks (in cases where separate application regions have separate linear systems, etc.), and a separate `Fmwk::Parameters` instance is associated with each of those solver-blocks. Each `Fmwk::Parameters` object is associated with a `Fmwk::LinearSystem` instance. This way if several LinearSystem instances are being used, they may all use the Aztec library and yet they may each specify different control parameters.

6.3 Supported Solver Libraries

The solver libraries available to SIERRA applications at the time of this writing are listed below. For the most part we won't detail the specific characteristics of each library, but references are provided for each one. Trilinos is somewhat of a special case, since it is a framework containing several packages that provide distinctly different services. These libraries predominantly provide iterative solution methods for sparse linear systems. The SPOOLES library provides a direct solution method.

- **FETI-DP** [16, 10]
- **HYPRE** [9]
- **ISIS++** [8]
- **PETSc** [6]
- **Prometheus** [1, 5]

- **Trilinos** [13, 14] Contained within Trilinos are several distinct packages which are listed below (note that this is not a complete list of Trilinos packages). Usage of these separate packages is mostly transparent to SIERRA users. If a Trilinos solver-block is included in the input file, the result is that matrix and vector data is assembled into Epetra objects, solution methods in the AztecOO package are used, along with preconditioners from the IFPACK package (if incomplete factorization is requested).
  - **AztecOO** Krylov solvers and various preconditioners.
  - **Amesos** Interfaces to direct solvers. Available to SIERRA in Trilinos versions 3.2.0 and later.
  - **Epetra** Matrix and Vector classes, and other data objects.
  - **IFPACK** Incomplete Factorization Package.
  - **ML** Multi-Level methods. Available to SIERRA in Trilinos versions 3.2.0 and later.
  - **NOX** Nonlinear solver package.
  - **TSF** Trilinos Solver Framework (system of abstract classes). This particular package may be used by the SIERRA framework in the future for generalizing interactions between linear solvers, nonlinear and eigensolvers, etc.

There is a library-specific implementation of `Fmwk::Solver_Support` for each of these solver libraries in the SIERRA framework as shown in figure 4. These classes provide code for parsing library-specific options from input files, as well as library-specific code for accessing solution methods, etc.

7 Miscellaneous Topics

Some SIERRA applications assemble and solve several linear systems per timestep, or solve an "aggregate" linear system where the matrix and right-hand-side may be linear combinations of other matrices and vectors. This section describes some aspects of these cases.

7.1 Sharing and Reusing Matrix, Matrix Graph

When several linear systems are solved per timestep, a couple of slightly different approaches are available. One approach is to only instantiate a single `Fmwk::LinearSystem` instance and repeatedly load and reset it, avoiding some costs such as memory use. This approach is not suitable for situations in which the separate linear systems are loaded by separate algorithms and application mechanics. In that case, it is better to use separate instances of `Fmwk::LinearSystem`, each with its own registered algorithms. However, it is possible to share a single set of underlying fei objects (`fei::MatrixGraph`, `fei::Matrix`, `fei::Vector`, `fei::LinearSystem`) among multiple instances of the `Fmwk::LinearSystem` class. This is accomplished via the methods on the `Fmwk::LinearSystem` class for getting and setting the matrix-graph and the linear-sytem container. There are a couple of limitations. Firstly, the structure (size, connectivities, etc.) of the linear systems being shared must be identical. Secondly, the `Fmwk::LinearSystem` instances must be bound

to the same underlying solver library (e.g., Trilinos, PETSc, etc.) otherwise the run-time types of the underlying data objects won't be the same. Once the method `Fmwk::LinearSystem`

7.2 Assembling Multiple Matrices

It is possible to assemble several matrices A_i and vectors b_j and then solve an aggregate linear system $Ax = b$ where

$$A = A_0 + A_1 + \cdots + A_n, \quad b = b_0 + b_1 + \cdots + b_m \tag{13}$$

and n is the number of matrices and m is the number of vectors. This is done using a single instance of `Fmwk::LinearSystem`, and carries the requirement that the individual matrices must have identical size and structure. If this feature is to be utilized, it must be specified when the `Fmwk::LinearSystem` object is constructed. Consult the doxygen-generated API documentation for the `Fmwk::LinearSystem` class.

8 Nonlinear Solvers, Eigensolvers

The SIERRA framework also supports the use of nonlinear solver and eigensolver packages, although these capabilities are somewhat less mature than the linear-solver services (meaning the framework interfaces to nonlinear and eigensolvers are still subject to change). In general the goal is to allow the same kind of flexibility with respect to choice of libraries as with linear-solver services. However, we currently only have one nonlinear solver library and one eigensolver, and so the abstraction layers are not yet completely specified.

8.1 `Fmwk::NonLinearSolver`

The `Fmwk::NonLinearSolver` class is the main point of interaction for applications using a nonlinear solver. See figure 5. It defines abstract interfaces which applications may implement in order to provide residual evaluations and preconditioning operations that get called during the solution process. `Fmwk::NonLinearSolver` also allows the application to register a `Fmwk::LinearSystem` instance. This allows for user-assembled preconditioning matrices to be used within the nonlinear solver. Currently the only nonlinear solver package available in SIERRA is the NOX package which resides within the Trilinos [14, 13] library.

8.2 EigenSolver

The eigensolver that is currently available in SIERRA is PARPACK [18, 15]. PARPACK is a library written in Fortran77 that implements Arnoldi methods for solving eigenvalue problems.

There is currently not a well-developed abstraction for eigensolvers in the SIERRA framework. The current approach for using the eigensolver is for an application to assemble a matrix using

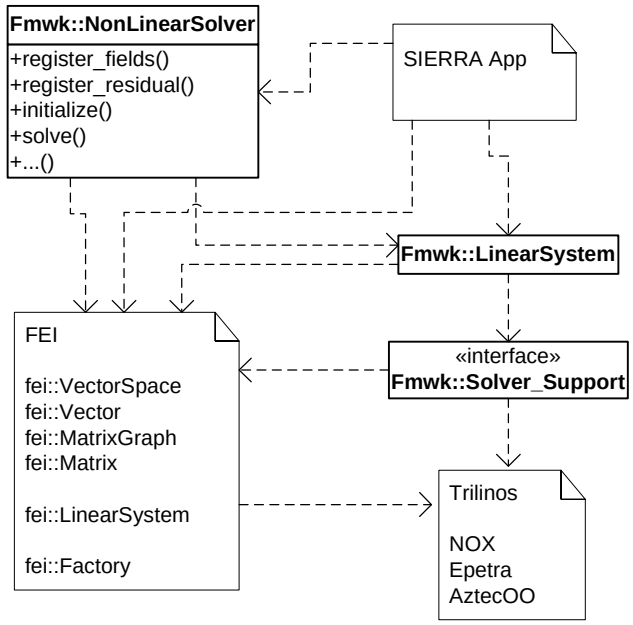


Figure 5. Interactions between `Fmwk::NonLinearSolver` and other entities.

the `Fmwk::LinearSystem` class, and then use the class `Solv_EigenSolver_ARPACK`, which provides methods that take fei::Matrix objects and return eigenvalues and eigenvectors. The eigenvectors are in the form of `fei::Vector` objects. The `Solv_EigenSolver_ARPACK` class creates the `fei::Vector` objects using an `fei::Factory` which must be supplied by the application (it can be obtained from the `Fmwk::LinearSystem` instance).

SIERRA's interface to PARPACK will soon be replaced by an interface to the Trilinos package Anasazi. As of this writing (June 2004), that interface has not yet been developed.

References

[1] Prometheus web site. `http://www.cs.berkeley.edu/~madams/Prom_intro.html`.

[2] Sand report example web page with notes and example files. `http://www.cs.sandia.gov/~rolf/SANDreport/index.html`.

[3] Sierra's internal web page. http://www.engsci.sandia.gov/sierra/sierraweb/fem/index.html.

[4] Sierra's new web site. http://sierra.sandia.gov.

[5] M. F. Adams. *Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations*. PhD thesis, University of California, Berkeley, 1998. Tech. Report UCB//CSD-99-1033.

[6] S. Balay, K. Buschelman, W. Gropp, D Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. Petsc 2.0 users manual. Technical report ANL-95/11 - Revision 2.1.6, Argonne National Laboratory, August 2003.

[7] R. Clay, K. Mish, I. Otero, L. Taylor, and A. Williams. An annotated reference guide to the finite element interface specification version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 1999.

[8] R. Clay, K. Mish, and A. Williams. Isis++ reference guide version 1.0. Technical report SAND97-8535, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 1997.

[9] R.D. Falgout and U. M. Yang. Hypre: A library of high performance preconditioners. Technical Report UCRL-JC-146175, Lawrence Livermore National Laboratory, Livermore, California 94550, 2002.

[10] C. Farhat and K. Pierson. The second generation of feti methods and their application to the parallel solution of large-scale linear and geometrically nonlinear structural analysis problems. *Computer Methods in Applied Mechanics and Engineering*, 184:333–374, 2000.

[11] R. Freund and N. Nachtigal. Qmr: A quasi-minimal residual method for non-hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins, 1996.

[13] M. A. Heroux et al. An overview of trilinos. Technical report SAND2003-2927, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, August 2003.

[14] M. A. Heroux and J. M. Willenbring. Trilinos users guide. Technical report SAND2003-2952, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, August 2003.

[15] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[16] M. Lesoinne and K. Pierson. Feti-dp: An efficient, scalable and unified dual-primal feti method. *12th International Conference on Domain Decomposition Methods*, 2001.

[17] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[18] Kristi Maschhoff and Danny Sorensen. A portable implementation of arpack for distributed memory parallel architectures. *Preliminary proceedings, Copper Mountain Conference on Iterative Methods*, 1996.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, 1996.

[20] Y. Saad and M.H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific and Stat. Comp.*, 7:856–869, 1986.

[21] P. Saint-Georges, Y. Notay, and G. Warzee. Efficient iterative solution of constrained finite element analyses. *Comput. Methods Appl. Mech. Engrg.*, 160:101–114, 1998.

[22] A. Williams. Finite element interface to linear solvers (fei) 2.9: Users guide and reference manual. Technical report SANDXX-XXXX, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2004.

A Parser Support and Class Instantiation

This appendix describes the 'plugging-in' of parser handlers for the solver support classes, and the creation of `Fmwk::LinearSystem` instances.

Getting solver support up and running in an application requires a couple of steps which are described here.

1. Start the solver-support's parser handler. In an application's 'main', or in some function that will be executed before the '`run_sierra()`' function is called, the appropriate solver-support class' factory method needs to be called to instantiate the class and activate its parser handler. Example:

   ```
   Solv_Trilinos_Support::factory();
   ```

   (These solver-support classes are singleton classes. Several different solver-support factories may be started, but only one of each.) Starting the Trilinos-support factory allows for Trilinos solver-blocks to be parsed from the input file:

   ```
   Begin Trilinos EQUATION SOLVER some_solver_name
   $ set some options and control parameters...
   End Trilinos EQUATION SOLVER some_solver_name
   ```

   Input-file solver-blocks are the mechanism by which application users specify parameters that control solver behavior, such as iteration limits, residual tolerances, preconditioner choices, etc.

2. The application region's parser code should include a handler that will construct an instance of `Fmwk::LinearSystem` in response to an input-file directive such as:

   ```
   USE LINEAR SOLVER some_solver_name
   ```

   The region's parser handler will have access to the string "some_solver_name", and this must be passed as a constructor argument to `Fmwk::LinearSystem`. The constructor uses this string to obtain the correct container of parsed solver options, and couples it with the `Fmwk::LinearSystem` instance along with a pointer to the `Solv_Trilinos_Support` object. The `Fmwk::LinearSystem` constructor takes a second string argument which will be used as the name of the `Fmwk::LinearSystem` instance. The reason for having two string arguments is to allow for the scenario where multiple `Fmwk::LinearSystem` instances are constructed sharing a single set of solver parameters.

B Parsing Solver Options

This appendix describes the connections between XML files and C++ source code that enable input-file commands to be translated into library specific option values to be passed to a linear-solver library. Consider an example. The command

```
Begin Trilinos EQUATION SOLVER flow_solver
 solution method = cg
End Trilinos EQUATION SOLVER flow_solver
```

in an input file specifies that the Conjugate Gradient method should be used with the Trilinos equation solver instance "flow_solver". This command must result in the pair of values "AZ_solver,AZ_cg" (which are macros) being passed to the AztecOO solver in the Trilinos library.

Input file equation solver commands such as this are defined in XML files that reside in the parser directory of the SIERRA framework repository. In particular, the file parser/apublic/Apub_SolvGeneric.xml defines line-commands such as "SOLUTION METHOD", along with enumerations for valid values such as "CG", "GMRES" and other possible solution methods. When the SIERRA application runs, its parser code reads this command and passes the corresponding enumerated values for "SOLUTION METHOD" and "CG" to the Trilinos parser handler (method on the class Solv_Trilinos_Support), provided that the Trilinos parser support singleton has been activated within the application. Unfortunately, the enumerations defined in XML files can't be directly used in C++ source code. For instance, the XML file enumeration for "SOLUTION METHOD" is 101, and the enumeration for "CG" is 0. So the Trilinos parser handler receives the pair of integer values 101 and 0. A corresponding set of enumerations usable in C++ source code is defined in the header include/solver/Solv_Support_Enums.h, and for instance, 101 is mapped to the name solv_solution_method and 0 is mapped to the name solver_cg. This way the Trilinos parser handler can be written in terms of the symbol "solv_solution_method" instead of the cryptic "101". Obviously the need to maintain dual sets of enumeration mappings causes the potential for errors, so care must be taken when making updates or changes.

Finally, the parser handler in Solv_Trilinos_Support maps the pair of symbols "solv_solution_method,solver_cg" to the Aztec-defined pair of macros "AZ_solver,AZ_cg" in preparation for setting up the AztecOO solution object.

Not all of the line-commands defined in Apub_SolvGeneric.xml are valid for all of the different solver libraries. To handle this, the block-command for each library has its own XML file which declares which of the generic solver line-commands are valid. For example, the file parser/solver/Solv_Trilinos.xml declares which line-commands may be used inside "Begin Trilinos EQUATION SOLVER" blocks.

C Dependencies and Third-Party-Library Management

C.1 Introduction

The SIERRA framework makes use of a variety of third-party libraries for solution of systems of linear equations, and other tasks. The dependence of the framework on third-party libraries gives rise to some software construction issues that must be resolved when altering or updating any of the libraries, or the framework.

There are two kinds of dependencies, namely compile-time and link-time dependencies. When a library is altered or updated, the updates are not made available to dependent executables unless some combination of re-linking and re-compiling is done. Examples of link-time dependencies include the BLAS and LAPACK libraries. To switch an executable from one BLAS library to another, it is only necessary to re-link the executable. Compile-time dependencies occur when a header from a library is used by code outside that library. If the header is changed during a library update, then dependent code must be re-compiled before executables are re-linked.

Performing a correct build in the SIERRA framework is complicated by the fact that dependencies exist not only between the framework and libraries, but also among the libraries themselves. The tools used to build SIERRA products make use of a system of XML files which express the dependence of a product on other products and on third-party libraries. Furthermore, XML files located within each third-party-library's source directory specify that library's name and version, as well as its dependence on other libraries. Since there are many products and libraries, an application can have a complex dependency graph, including 'diamond-shaped' dependence where it depends

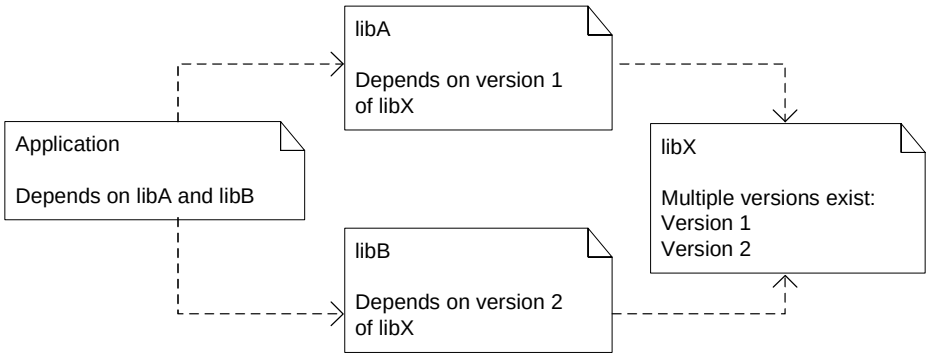


Figure C.1. Conflict in 'diamond-shaped' dependency

on two separate libraries, each of which in turn depend on a third library. Each dependence specifies both a name and a version, so the tools can detect situations where, for instance, different versions of a library appear in a dependency graph, causing a conflict for an application as is shown in figure C.1. Note that if the situation illustrated in figure C.1 occurs, then the application is in a 'broken' state because the tools will not allow a build to complete if there is a conflict in the dependencies.

This note specifically addresses the dependencies between the SIERRA framework, the FEI

library, and third-party solver libraries.

C.2 Framework, FEI and solver libraries

The SIERRA framework uses several different solver libraries, and linear-system assembly is managed through abstractions provided by the FEI library. The way this is accomplished, is as follows. The FEI library publishes interfaces for SIERRA to use, in FEI.h and other headers. The SIERRA framework passes linear-system data to the FEI interface, and the FEI implementation relays the data to either of two (internally defined/used) abstract interfaces named LinearSystemCore and FiniteElementData. These interfaces provide the mechanism for coupling the FEI implementation with an underlying solver-library, so for each solver library that is to be used with the FEI, there must be a solver-specific implementation of either LinearSystemCore or FiniteElementData. This way any one of several solver libraries may be coupled with the FEI implementation at run-time.

Figure C.2 shows the dependencies among the SIERRA framework, the FEI and the solver-

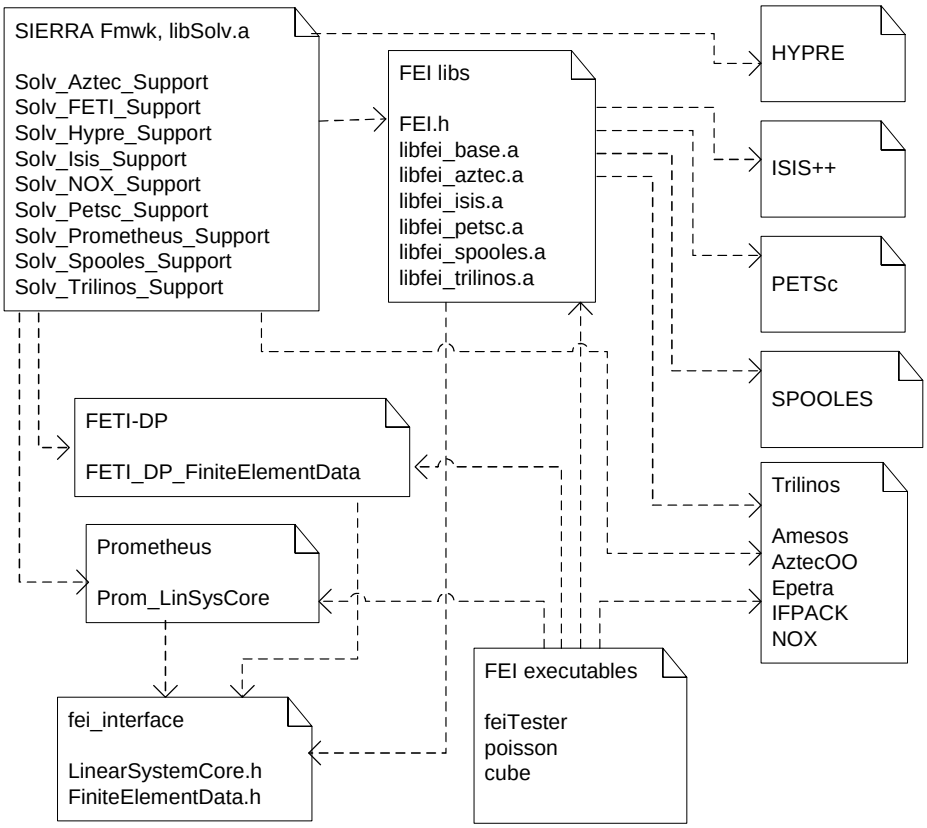


Figure C.2. SIERRA, FEI, and solver dependencies

libraries. The LinearSystemCore.h and FiniteElementData.h headers reside in their own ”library”, which is called fei_interface. FEI ”libs” and ”executables” are shown as separate entities even though they reside in a single repository. This is because they have separate XML files, so that the SIERRA

framework can depend on ”FEI libs” without inheriting all dependencies downstream of ”FEI executables”.

The SIERRA framework has direct dependencies on FETI-DP and Prometheus, but not on most other solver libraries (such as SPOOLES, etc.). This is due to the location of the solver-specific LinearSystemCore implementation or FiniteElementData implementation. FETI-DP and Prometheus contain their own implementations of these interfaces. In the case of SPOOLES and several other libraries, the LinearSystemCore implementation resides in the FEI library. This is due to the fact that those implementations were written by us, rather than by the library authors themselves.

The headers that define the LinearSystemCore and FiniteElementData interfaces rarely need to be changed. But occasionally a change is necessary and a series of coordinated updates is required to propagate the change up to the SIERRA framework.

In the SIERRA system there is a policy that installed libraries are not to be altered. If a library needs to be altered or updated in any way, it must be done by installing a new version of the library containing the update, and then switch all dependent libraries, products and applications from the old version to the new version.

At this point it is worth noting that some of the third-party libraries used by SIERRA are developed within the SIERRA tools. Releases of these libraries are created by branching the repository and labeling the branch with a version. But the branch may continue to be modified by checkins, at the developer’s discretion (technically violating the above-mentioned policy against altering installed libraries). The FEI library and the FETI-DP library are both developed within the SIERRA tools. The Prometheus library is treated as a ”true” third-party library. It is developed elsewhere and new versions are imported as tar-files, unpacked, installed and then write-protected, thereby preventing subsequent modification.

In the case of modifying a header in the ”fei_interface” product, a series of updates is required, including re-compiling the FEI library as well as the Prometheus and FETI-DP libraries, before finally re-building the SIERRA framework. As soon as any of these updates is done, the SIERRA framweork is in a broken state until all of the updates are done. Since some of the libraries are developed under the SIERRA tools, it is tempting to make changes like this by simply performing a checkin and modifying the appropriate library in place. However, in the case of a header change in an fei_interface header, this causes mis-matched header errors in upstream libraries (Prometheus, FEI and FETI-DP) until all have been re-compiled. The only way to do the updates without temporarily breaking dependent libraries, is by creating a new version of each and finally switching SIERRA to all of those new versions at once.

In some cases it has been judged preferable to change libraries in place and accept the fact that other products are broken temporarily, rather than go through the more onerous process of creating the series of new library versions and then switching the SIERRA XML files. If done late in the day, it may be possible to complete the updates and restore the system to working order without impacting other developers. However, this is not advisable. Another policy in SIERRA is that no change should be committed without running tests for dependent products to verify that the change doesn’t have any negative side-effects. If libraries are changed in-place, it is difficult to detect and correct negative side-effects before other developers are impacted. The only safe and correct approach is to create new versions of libraries to be updated, so that dependent products can be tested against the new versions before XML changes are committed and the new version is thrust

upon the rest of the developer community.

Distribution:

Internal:

1	MS 0384	T.C. Bickel	9100
1	MS 9003	K.E. Washington	8900
1	MS 0384	H.S. Morgan	9140
1	MS 0382	J.R. Stewart	9143
1	MS 0382	E.A. Boucheron	9141
1	MS 0380	K.F. Alvin	9142
1	MS 0823	J.D. Zepper	9324
1	MS 1110	D.E. Womble	9214
1	MS 9917	S.W. Thomas	8962
1	MS 9915	M.L. Koszykowski	8961

1	MS 0382	H.C. Edwards	9143
1	MS 0382	K.D. Copps	9143
1	MS 0382	G.D. Sjaardema	9143
1	MS 0382	J.R. Overfelt	9143
1	MS 0382	K.N. Belcourt	9143
1	MS 0382	K.M. Aragon	9143
1	MS 0382	D.M. Brethauer	9143
1	MS 0382	M.E. Hamilton	9143
10	MS 0382	A.B. Williams	9143

1	MS 0382	S.W. Bova	9141
1	MS 0382	S.P. Domino	9141
1	MS 0382	T.O. Okusanya	9141
1	MS 0382	C.K. Newman	9141
1	MS 0382	R.R. Lober	9141
1	MS 0382	A.A. Lorber	9141
1	MS 0382	S.R. Subia	9141
1	MS 0380	J.D. Hales	9142
1	MS 0380	K.H. Pierson	9142
1	MS 0380	M.K. Bhardwaj	9142
1	MS 0380	G.M. Reese	9142
1	MS 0834	M.M. Hopkins	9114
1	MS 0382	P.K. Notz	9114

1	MS 9159	M.F. Adams	9214
1	MS 1110	M.A. Heroux	9214
1	MS 0316	R. Hooper	9233
1	MS 0316	J.N. Shadid	9233
1	MS 0316	T.M. Smith	9233
1	MS 0370	R.A. Bartlett	9211
1	MS 9018	Central Technical Files	8945-1
2	MS 0899	Technical Library	9616

External:

Evi Dube
Lawrence Livermore National Laboratory
Livermore, CA 94551-0808